US006041179A

United States Patent [19]

Bacon et al.

[11] **Patent Number: 6,041,179**

[45] **Date of Patent: Mar. 21, 2000**

[54] **OBJECT ORIENTED DISPATCH OPTIMIZATION**

[75] Inventors: **David Francis Bacon**, New York; **Mark N. Wegman**; **Frank Kenneth Zadeck**, both of Ossining, all of N.Y.

[73] Assignee: **International Business Machines Corporation**, Armonk, N.Y.

IBM

[21] Appl. No.: **08/723,058**

[22] Filed: **Oct. 3, 1996**

[51] **Int. Cl.**^{7} **G06F 13/00**

[52] **U.S. Cl.** **395/709**

[58] **Field of Search** 395/703, 702, 395/700, 685, 683, 650, 707, 709

[56] **References Cited**

U.S. PATENT DOCUMENTS

5,450,583	9/1995	Inada	395/650
5,499,371	3/1996	Henninger et al.	395/700
5,579,518	11/1996	Yasumatsu	395/705
5,600,838	2/1997	Guillen et al.	395/683
5,659,751	8/1997	Heninger	395/685
5,675,801	10/1997	Lindsey	395/702

OTHER PUBLICATIONS

Amitabh Srivastava, "Unreachable Procedures in Object-Oriented Programming", ACM Letters on Programming Languages and Systems, vol. 1, No. 4, Dec., 1992, pp. 355–364.

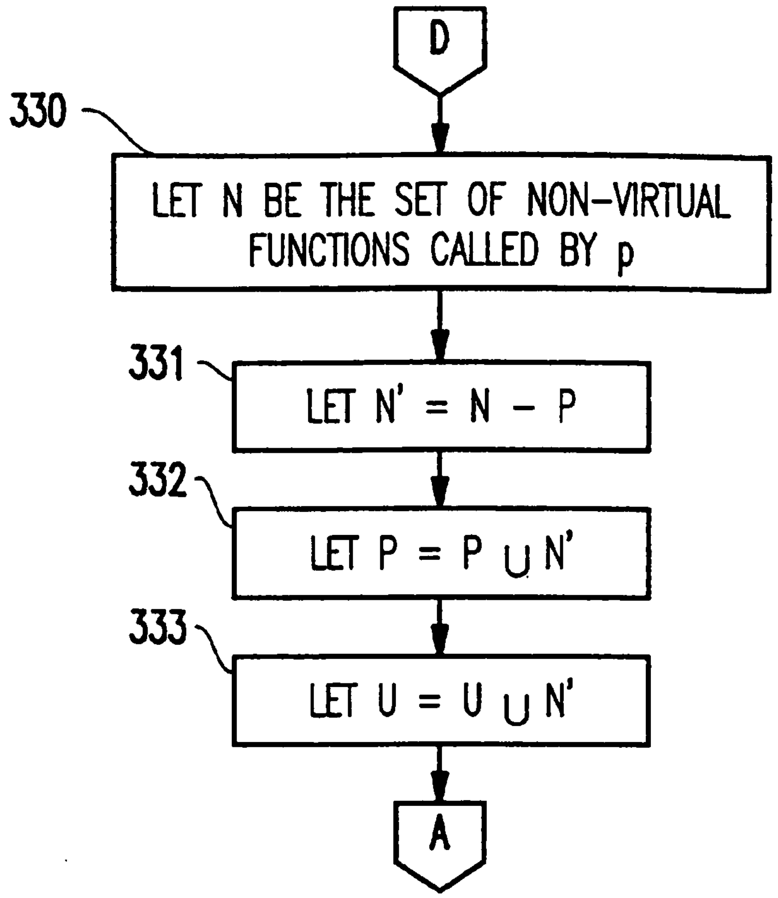
Brad Caler & Dirk Grunwald, "Reducing Indirect Function Call Overhead In C++ Programming", ACM Principles and Practice of Programming Languages, Portland, Oregon, 1994.

Primary Examiner—Emanuel Todd Voeltz
Assistant Examiner—Matthew Smithers
Attorney, Agent, or Firm—Whitham, Curtis & Whitham; Louis J. Percello

[57] **ABSTRACT**

An object oriented dispatch optimization method determines statically which body of code will be executed when a method is dispatched. The program code is examined to identify all procedure bodies that can be invoked for a given class and a given method. An identified procedure body is analyzed to determine whether a method invocation on a pointer can invoke only one procedure body. Based on this analysis, either the procedure body or the invocation mechanism is changed so a unique procedure is directly called without a test or dispatch being used.

2 Claims, 5 Drawing Sheets



US-PAT-NO:	6041179
DOCUMENT-IDENTIFIER:	US 6041179 A
TITLE:	Object oriented dispatch optimization
DATE-ISSUED:	March 21, 2000

INVENTOR-INFORMATION:

NAME COUNTRY	CITY	STATE	ZIP CODE	
Bacon; David Francis	New York	NY	N/A	N/A
Wegman; Mark N.	Ossining	NY	N/A	N/A
Zadeck; Frank Kenneth	Ossining	NY	N/A	N/A

US-CL-CURRENT: 717/116, 717/151

ABSTRACT:

An object oriented dispatch optimization method determines statically which body of code will be executed when a method is dispatched. The program code is examined to identify all procedure bodies that can be invoked for a given class and a given method. An identified procedure body is analyzed to determine whether a method invocation on a pointer can invoke only one procedure body. Based on this analysis, either the procedure body or the invocation mechanism is changed so a unique procedure is **directly** called without a test or dispatch being used.

2 Claims, 5 Drawing figures

Exemplary Claim Number: 2

Number of Drawing Sheets: 5

---------- KWIC ---------

Abstract Text - ABTX (1):

An object oriented dispatch optimization method determines statically which body of code will be executed when a method is dispatched. The program code is examined to identify all procedure bodies that can be invoked for a given class and a given method. An identified procedure body is analyzed to determine whether a method invocation on a pointer can invoke only one procedure body. Based on this analysis, either the procedure body or the invocation mechanism is changed so a unique procedure is **directly** called without a test or dispatch being used.

Brief Summary Text - BSTX (14):

In Smalltalk, all method dispatches are made dynamically. In other words, the actual method body that will be invoked at a particular method invocation site is determined by a combination of the class of the object and the name of the method. This may be done by following a pointer from the object to its class, and then looking up the method in the classes' method table. In contrast, C++ allows the programmer to decide whether a particular method should be dispatched statically or dynamically. Methods that are dispatched dynamically are called "virtual methods." The present invention is concerned only with virtual method invocations and, therefore, methods that are dispatched statically will not be described. Accordingly, when the following description references C++ programs, the terms "method dispatch" or "method invocation," unless otherwise qualified, shall mean "virtual method dispatch" or "virtual method invocation." Note that in the C++ literature, a virtual method dispatch is also referred to as a "**virtual function call.**"

Brief Summary Text - BSTX (15):
Method dispatches are a major source of complexity when trying to optimize the programs. More particularly, there is a **direct** cost associated with method dispatches in the form of the extra instructions required for the dynamic dispatch, including extra memory operations, and pipeline penalties caused by branching to an unknown address. For C++ programs, these costs have been estimated as ranging from 0% of the total run time, for programs that make no significant use of method dispatch, to 6%, for programs that make moderate use of method dispatch, to 27%, for programs that make extensive use of method dispatch.

Brief Summary Text - BSTX (18):
There are methods in the relevant art **directed** to reducing, at least partially, the above-identified problems relating to inlining and compilation speed and compiled code size, but each has limitations in performance, or requires so large of a processing time as to be impracticable.

Brief Summary Text - BSTX (20):
It can be seen that if one can determine that for all dispatches from a particular point in the program a particular method body must be invoked, then that dispatch can be replaced by a **direct** call to the appropriate body of code without modifying the end result of the program, and hence this replacement can safely be done automatically. Calder and Grunwald have published some preliminary measurements stating that, in their selected set of particular benchmark programs, it is possible with the Resolution by Unique Name Method to replace approximately one third of the method dispatches by a **direct** call.

Brief Summary Text - BSTX (25):
determining whether that method is one of the methods previously determined to have only one function and, if so, converting the dispatch to a **direct** call.

Brief Summary Text - BSTX (26):
There is at least one problem with the Resolution by Unique Name Method, though, which is that the replacement of a dispatch by a **direct** call cannot be done by a traditional compiler. The major reason is that a compiler is usually not given an entire program at a time, but only a piece of the program. The entire program is composed by a program called a linker. Hence, the Resolution by Unique Name Method is generally referred to as a link-time optimization. It is possible for Calder and Grunwald's method to be performed at other times, if it is given appropriate information. For example, if a database of information about an entire program is built first, then when a compiler is used to compile a piece of a program, it can refer to information in the built database about the rest of the program. However, if the rest of the program changes, and the information which was relied upon changes, the program may have to be recompiled using the new information.

Brief Summary Text - BSTX (27):
Another related method **directed** to resolving virtual function dispatch is called class hierarchy analysis (henceforth "CHA"), and is described by Dean, Grove and Chambers in "Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis", in the Proceedings of the Ninth European Conference

on Object-Oriented Programming, Springer-Verlag, 1995. CHA uses the type information supplied only in statically typed languages, therefore, is not applicable to a dynamically-typed OOP languages such as Smalltalk.

Brief Summary Text - BSTX (28):
The meaning of statically-typed, which relates to OOP languages such as C++ is as follows: When a **virtual function call** is made in C++, it is dispatched through a pointer or a reference to an object, and that pointer has a particular type. The type is either specified explicitly, i.e., statically, by the programmer, or is statically derivable by the compiler from other type information, as shown in the example below:

Brief Summary Text - BSTX (33):
It can be seen by one of ordinary skill that the set of **virtual call** sites resolved by CHA is a superset of the call sites resolved by Calder and Grunwald's Unique Name method. This is because CHA starts by using the signature of the function and then uses the static type information to identify additional **virtual call** sites.

Brief Summary Text - BSTX (39):
As will be seen from the description below, the present invention, which will be called the Rapid Type Analysis, (or "RTA"), converts many cases of method dispatches to **direct** calls which would not have been converted by the class hierarchy analysis (CHA). Further, the present invention can potentially reduce the set of possible method bodies that can be invoked at a method dispatch site. The RTA method accomplishes this function by inspecting each function call only once, instead of the repeated inspections required by alias analysis.

Brief Summary Text - BSTX (40):
More particularly, the present invention provides a process for increasing the execution speed of programs which comprises the steps of examining a program to identify all procedure bodies that can be invoked for a given class and a given method, analyzing a procedure body to determine whether a method invocation on a pointer can invoke only one procedure body, and changing either the procedure body or the invocation mechanism so that a unique procedure is **directly** called without a test or dispatch being used.

Brief Summary Text - BSTX (45):
Code **directly** invoked from live code is flagged as live. This process is iterated until there is no more live code discovered. All other code is now known to be dead and can be safely eliminated. As will be described, "optimistic" analysis can obtain a better result. This can be seen by considering a routine that invokes itself but is not invoked from anywhere else. The optimistic analysis will correctly determine that the code is, in fact, dead, whereas the more conservative analysis will not.

Brief Summary Text - BSTX (46):
The second detection identifies classes and derived classes for which there are no allocations of objects. It is then known that all methods defined for that class cannot be invoked. Subsequently it is known that certain code, i.e., the code relating to these methods, is dead. Next, since it is then known that the only allocations for certain classes are within that dead code, this second detection can be iterated. The present inventive method combines these two detection mechanisms for discovering dead code. As will be described, the present invention preferably employs techniques of optimistic analysis to obtain a stronger result. Further, in a preferred implementation, the program code is examined to identify all method bodies that can be invoked

for a given class and a given method. An identified procedure body is analyzed to determine whether a method invocation on a pointer can invoke only one method body. Based on this analysis, either the method body or the invocation mechanism is changed so that a unique method is **directly** called without a test or dispatch being used. More specifically, rather than using the alias analysis method of trying to identify the possible types at each **virtual function call** site, the methods implemented in this invention are **directed** to identifying the set of possible types that can be created by a program. This greatly simplifies the task. Because of the design and usage conventions of C++ classes, the process is almost as precise as more complex alias analysis methods that attempt to track the possible type of each individual variable. The result is a reduced processing time, which can be as much as fifty to one thousand times, depending on the particular code. Further, the present method is also easily adapted to allow separate compilation.

Detailed Description Text - DETX (11):
The set of virtual function invocations I is also initialized to be empty (303). The set I records, in a language-dependent manner, the kinds of **virtual function calls** that have been made by procedures that have been determined to be live by the process. Each member of the set I is a pair of <function-name, type-information>. For Smalltalk, the type information is simply the arity (number of arguments) of the function called. For C++ and Java, the type information is the declared static type of the pointer or reference through which the **virtual call** was made, and the types of the arguments of the function.

Detailed Description Text - DETX (13):
The set Cp of classes created by procedure p is computed in a language dependent manner (306). For statically typed languages, this information is easily available through a simple static analysis of the procedure body, since when a new class object is created the class type must be specified **directly**. For dynamically typed languages like Smalltalk, calculating Cp is more complex because the argument to the NEW operation may be a type variable, not just a type constant. For Smalltalk, Cp can either be the set of classes referred to in the procedure body, or else a pre-analysis will have to take place which calculates the set of possible types that could be created.

Detailed Description Text - DETX (17):
Otherwise, a language-dependent test is performed as to whether the method m is compatible with any of the live **virtual call** invocations in I (318). For Smalltalk, this test simply involves determining whether there is an element <function-name,arity> contained in I, where function-name is the name of m and arity is the number of its arguments. For C++ and Java, the test is more complex: is there a member of I with the same name and argument types as m, whose statically declared type is either c (the class type of method m) or a derived class of c? If this test fails, then there is no method invocation in the live procedures P which is compatible with m, so control returns to 314 and the next method of c is examined. If the test succeeds, then m is a new live method, and is added to both P and U (319); control then returns to 315 and the next method of c is examined.

Detailed Description Text - DETX (18):
When all the classes Cp instantiated by procedure p have been examined (310) control passes to C in FIG. 2(c). This is the second phase of the algorithm, which handles the **virtual call** sites in procedure p. The first phase, shown in FIG. 2(b), handled the new classes created by procedure p.

Detailed Description Text - DETX (28):
In an example implementation, the invention is implemented within a compiler

for the C++ language. The compiler builds a **call graph** for the program, and the sets P and U are represented implicitly by boolean variables attached to each procedure object. Similarly, there is a graph which represents the class hierarchy, and the set C of live classes is represented implicitly by a boolean value attached to each class object. The set I of call instances is represented by a hash table which is indexed by function signature.

Detailed Description Text - DETX (31):
The set Ip of **virtual calls** by procedure p is also pre-computed by an earlier compiler phase, and the iteration over this set is implemented by traversal of the linked list.

Detailed Description Text - DETX (32):
The set T of methods of live classes that are compatible with a **virtual call** instance i is computed by our example implementation by finding the signature of i with its associated class in the class hierarchy graph, and recursively looking up the same signature in all of the derived classes represented in the class hierarchy graph.

Detailed Description Text - DETX (35):
1. Resolution of Dynamic Dispatch. A **virtual function call** can be converted to a **direct** function call when there is only one method in P, the set of live procedures, that is compatible with the function name and type information of the called function. If there is more than one compatible function, but only a small number, the call can be resolved by introducing explicit tests for the reduced number of possible target functions.

Detailed Description Text - DETX (37):
3. Programming Environment. A programming environment can provide a view of the program in which only those procedures that are live (those in P) are displayed to the programmer, and only those procedures in P are listed or displayed as potential targets of **virtual function calls**. Classes not in C, the set of live classes, can also be elided. This elision of information will allow the programmer to concentrate on the functional parts of the program.

Claims Text - CLTX (6):
converting indirect methods invocations to **direct** method calls which do not perform a runtime type test or dynamic dispatch; and